(51) International Patent Classification ⁶: G06F 12/02	A1	(11) International Publication Number: WO 99/00731
		(43) International Publication Date: 7 January 1999 (07.01.99)

(21) International Application Number: PCT/US98/13563

(22) International Filing Date: 29 June 1998 (29.06.98)

(30) Priority Data:
08/884,377   27 June 1997 (27.06.97)   US

(63) Related by Continuation (CON) or Continuation-in-Part
(CIP) to Earlier Application
US   08/884,377 (CON)
Filed on   27 June 1997 (27.06.97)

(71) Applicant (for all designated States except US): CHAMELEON SYSTEMS, INC. [US/US]; Suite 275, 960 N. San Antonio Road, Los Altos, CA 94022 (US).

(72) Inventors; and
(75) Inventors/Applicants (for US only): COOKE, Laurence, H. [US/US]; 25399 Spanish Ranch Road, Los Gatos, CA 95030 (US). PHILIPS, Christopher, E. [US/US]; 5888 Assis Court, San Jose, CA 95138 (US). WONG, Dale [US/US]; 855 35th Avenue, San Francisco, CA 94121 (US).

(74) Agent: KREBS, Robert, E.; Burns, Doane, Swecker & Mathis, L.L.P., P.O. Box 1404, Alexandria, VA 22313–1404 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: METHOD FOR COMPILING HIGH LEVEL PROGRAMMING LANGUAGES

(57) Abstract

A computer program (item 101), written in a high level programming language, is compiled (item 103) into an intermediate data structure (105) which represents its control and data flow. This data structure is analyzed (item 111) to identify critical blocks of logic which can be implemented as an application specific integrated circuit (item 117) to improve the overall performance. The critical blocks of logic are first transformed into new equivalent logic with maximum data parallelism. The new parallelized logic is then translated into a Boolean gate representation which is suitable for implementation on an application specific integrated circuit (item 117). The application specific integrated circuit (item 117) is coupled with a generic microprocessor via custom instructions for the microprocessor (item 107). The original computer program is then compiled into object code (item 109) with the new expanded target instruction set.
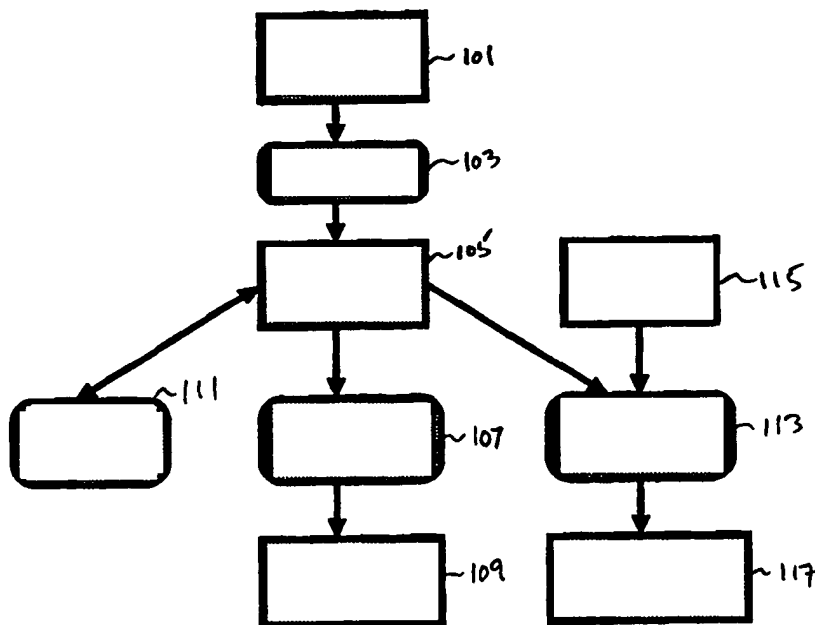
101...PROGRAMMING LANGUAGE SOURCE CODE
103...SOURCE CODE COMPILER
105...COMPILED CODE INTERMEDIATE FORMAT
107...CODE GENERATOR FOR MICROPROCESSOR
109...OBJECT CODE FOR MICROPROCESSOR
111...OPTIMIZER AND PARALLELIZER
115...CIRCUIT PRIMITIVES LIBRARY
113...SILICON COMPILER FOR INTEGRATED CIRCUIT
117...APPLICATION SPECIFIC INTEGRATED CIRCUIT

METHOD FOR COMPILING HIGH LEVEL PROGRAMMING LANGUAGES

5   BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to reconfigurable computing.

10   2. State of the Art

Traditionally, an integrated circuit must be designed by describing its structure with circuit primitives such as Boolean gates and registers. The circuit designer must begin with a specific application in mind, e.g. a video compression algorithm, and the resulting integrated circuit can only be used for the targeted application.

15   Alternatively, an integrated circuit may be designed as a general purpose microprocessor with a fixed instruction set, e.g. the Intel x86 processors. This allows flexibility in writing computer programs which can invoke arbitrary sequences of the microprocessor instructions. While this approach increases the flexibility, it decreases the performance since the circuitry cannot be optimized for any specific application.

20   It would be desirable for high level programmers to be able to write arbitrary computer programs and have them automatically translated into fast application specific integrated circuits. However, currently there is no bridge between the computer programmers, who have expertise in programming languages for microprocessors, and the application specific integrated circuits, which require expertise in circuit design.

25   Research and development in integrated circuit design is attempting to push the level of circuit description to increasingly higher levels of abstraction. The current state of the art is the "behavioral synthesizer" whose input is a behavioral language description of the circuit's register/transfer behavior and whose output is a structural description of the circuit elements required to implement that behavior. The input

30   description must have targeted a specific application and must describe its behavior in high level circuit primitives, but the behavioral compiler will automatically determine

how many low level circuit primitives are required, how these primitives will be shared

between different blocks of logic, and how the use of these primitives will be

scheduled. The output description of these circuit primitives is then passed down to a

"logic synthesizer" which maps the circuit primitives onto a library of available "cells",

5   where each cell is the complete implementation of a circuit primitive on an integrated

circuit. The output of the logic synthesizer is a description of all the required cells and

their interconnections. This description is then passed down to a "placer and router"

which determines the detailed layout of all the cells and interconnections on the

integrated circuit.

10   On the other hand, research and development in computer programming is also

attempting to push down a level of abstraction by matching the specific application

programs with custom targeted hardware. One such attempt is the Intel MMX

instruction set. This instruction set was designed specifically to accelerate applications

with digital signal processing algorithms. Such applications may be written generically

15   and an MMX aware compiler will automatically accelerate the compiled code by using

the special instructions. Another attempt to match the application with appropriate

hardware is the work on parallelizing compilers. These compilers will take a computer

program written in a sequential programming language and automatically extract the

implicit parallelism which can then be targeted for execution on a variable number of

20   processors. Thus different applications may execute on a different number of

processors, depending on their particular needs.

Despite the above efforts by both the hardware and software communities, the

gap has not yet been bridged between high level programming languages and integrated

circuit behavioral descriptions.

25

SUMMARY OF THE INVENTION

A computer program, written in a high level programming language, is

compiled into an intermediate data structure which represents its control and data flow.

This data structure is analyzed to identify critical blocks of logic which can be

30   implemented as an application specific integrated circuit to improve the overall

-2-

performance. The critical blocks of logic are first transformed into new equivalent logic with maximal data parallelism. The new parallelized logic is then translated into a Boolean gate representation which is suitable for implementation on an application specific integrated circuit. The application specific integrated circuit is coupled with a

5   generic microprocessor via custom instructions for the microprocessor. The original computer program is then compiled into object code with the new expanded target instruction set.

In accordance with one embodiment of the invention, a computer implemented method automatically compiles a computer program written in a high level

10   programming language into a program for execution by one or more application specific integrated circuits coupled with a microprocessor. Code blocks the functions of which are to be performed by circuitry within the one or more application specific integrated circuits are selected, and the code blocks are grouped into groups based on at least one of an area constraint and an execution timing constraint. Loading and

15   activation of the functions are scheduled; and code is produced for execution by the microprocessor, including instructions for loading and activating the functions.

In accordance another aspect of the invention, a computer implemented method automatically compiles a computer program written in a high level programming language into one or more application specific integrated circuits. In accordance with

20   yet another aspect of the invention, a computer implemented method automatically compiles a computer program written in a high level programming language into one or more application specific integrated circuits coupled with a standard microprocessor. In accordance with still another aspect of the invention, a reconfigurable logic block is locked by compiled instructions, wherein an activate configuration instruction locks the

25   block from any subsequent activation and a release configuration instruction unlocks the block. In accordance with a further aspect of the invention, a high level programming language compiler automatically determines a set of one or more special instructions to extend the standard instruction set of a microprocessor which will result in a relative performance improvement for a given input computer program. In accordance with yet

30   a further aspect of the invention, a method is provided for transforming the execution

-3-

of more than one microprocessor standard instruction into the execution of a single

special instruction. In accordance with still a further aspect of the invention, a high

level programming language compiler is coupled with a behavioral synthesizer via a

data flow graph intermediate representation.

5

BRIEF DESCRIPTION OF THE DRAWING

The present invention may be further understood from the following description

in conjunction with the appended drawing. In the drawing:

Figure 1 shows the design methodology flow diagram of the preferred

10   embodiment of a compiler.

Figure 2 shows the control flow for the operation of the preferred embodiment

of an application specific integrated circuit.

Figure 3 shows a fragment of a high level source code example which can be

input into the compiler.

15   Figure 4 shows the microprocessor object code for the code example of Figure 3

which would be output by a standard compiler.

Figure 5 shows an example of the application specific circuitry which is output

by the compiler for the code example of Figure 3.

20   ## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

In accordance with the preferred embodiment of the present invention, a

method is presented for automatically compiling high level programming languages into

application specific integrated circuits (ASIC).

Referring to Figure 1, the computer program source code 101 is parsed with

25   standard compiler technology 103 into a language independent intermediate format 105.

The intermediate format 105 is a standard control and data flow graph, but with the

addition of constructs to capture loops, conditional statements, and array accesses. The

format's operators are language independent simple RISC-like instructions, but with

additional operators for array accesses and procedure calls. These constructs capture

30   all the high level information necessary for parallelization of the code. For further

description of a compiled intermediate format see for example S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. M. Murphy, R. S. French, M. S. Lam and M. W. Hall; Multiprocessors from a Software Perspective; IEEE Micro, June 1996; pages 52-61.

5   Because standard compiler technology is used, the input computer program can be any legal source code for a supported high level programming language. The methodology does not require a special language with constructs specifically for describing hardware implementation elements. Front end parsers currently exist for ANSI C and FORTRAN 77 and other languages can be supported simply by adding

10   new front end parsers. For further information on front end parsers see for example C. W. Fraser and D. R. Hanson; A Retargetable Compiler for ANSI C; SIGPLAN Notices, 26(10); October 1991.

From the intermediate format 105, the present methodology uniquely supports code generation for two different types of target hardware: standard microprocessor and

15   ASIC. Both targets are needed because while the ASIC is much faster than the microprocessor, it is also much larger and more expensive and therefore needs to be treated as a scarce resource. The compiler will estimate the performance versus area tradeoffs and automatically determine which code blocks should be targeted for a given available ASIC area.

20   Code generation for the microprocessor is handled by standard compiler technology 107. A code generator for the MIPS microprocessor currently exists and other microprocessors can be supported by simply adding new back end generators. In the generated object code 109, custom instructions are inserted which invoke the ASIC-implemented logic as special instructions.

25   The special instructions are in four general categories: load_configuration, activate_configuration, invoke_configuration, release_configuration. The load_configuration instruction identifies the address of a fixed bit stream which can configure the logic and interconnect for a single block of reconfigurable logic on the ASIC. Referring to Figure 2, the ASIC 200 may have one or more such blocks 201a,

30   201b on a single chip, possibly together with an embedded microprocessor 205 and

control logic 207 for the reconfigurable logic. The identified bit stream may reside in, for example, random access memory (RAM) or read-only-memory (PROM or EEPROM) 203. The bit stream is downloaded to a cache of possible block configurations on the ASIC. The activate_configuration instruction identifies a

5   previously downloaded configuration, restructures the reconfigurable logic on the ASIC block according to that configuration, and locks the block from any subsequent activate instructions. The invoke_configuration instruction loads the input operand registers, locks the output registers, and invokes the configured logic on the ASIC. After the ASIC loads the results into the instruction's output registers, it unlocks the registers and

10   the microprocessor can take the results and continue execution. The release_configuration instruction unlocks the ASIC block and makes it available for subsequent activate_configuration instructions. For further description of an embedded microprocessor with reconfigurable logic see U.S. Patent Application 08/884,380 of L. Cooke, C. Phillips, and D. Wong for An Integrated Processor and Programmable

15   Data Path Chip for Reconfigurable Computing, incorporated herein by reference.

Code generation for the ASIC logic can be implemented by several methods. One implementation passes the intermediate control and data flow graphs to a behavioral synthesis program. This interface could be accomplished either by passing the data structures directly or by generating an intermediate behavioral language

20   description. For further discussion of behavioral synthesis see for example D. Knapp; Behavioral Synthesis; Prentice Hall PTR; 1996. An alternative implementation generates one-to-one mappings of the intermediate format primitives onto a library of circuit implementations. For example: scalar variables and arrays are implemented as registers and register files with appropriate bit widths; arithmetic and Boolean operators

25   such as add, multiply, accumulate, and compare are implemented as single cells with appropriate bit widths; conditional branch implementations and loops are implemented as state machines. In general, as illustrated in Figure 1, a silicon compiler 113 receives as inputs compiled code in the intermediate format 105 and circuit primitives from a circuit primitive library 115 and produces layout or configuration information for an

30   ASIC 117. For further discussion of techniques for state machine synthesis see for

example G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti; Design Systems for VLSI Circuits; Martinus Nijhoff Publishers; 1987; pp. 327-364.

After the synthesis or mapping step is completed, an equivalent list of cells and their interconnections is generated. This list is commonly referred to as a netlist. This

5   netlist is then passed to a placer and router which determines the actual layout of the cells and their interconnections on an ASIC. The complete layout is then encoded and compressed in a bit stream format which can be stored and loaded as a single unit to configure the ASIC. A step-by-step example of the foregoing process is illustrated in Figure 3, Figure 4, and Figure 5. For a general discussion of place and route

10   algorithms see T. Ohtsuki; Layout Design and Verification; North-Holland; 1986; pp. 55-198.

The basic unit of code that would be targeted for an ASIC is a loop. A single loop in the input source code may be transformed in the intermediate format into multiple constructs for runtime optimization and parallelization by optimizer and

15   parallelizer 111 in Figure 1. The degree of loop transformation for parallel execution is a key factor in improving the performance of the ASIC versus a microprocessor. These transformations are handled by standard parallelizing compiler technology which includes constant propagation, forward propagation, induction variable detection, constant folding, scalar privatization analysis, loop interchange, skewing, and reversal.

20   For a general discussion of parallel compiler loop transformations see Michael Wolfe; High Performance Compilers for Parallel Computing; Addison-Wesley Publishing Company; 1996; pp. 307-363.

To determine which source code loops will yield the most relative performance improvement, the results of a standard source code profiler are input to the compiler.

25   The profiler analysis indicates the percentage of runtime spent in each block of code. By combining these percentages with the amount of possible parallelization for each

loop, a figure of merit can be estimated for the possible gain of each loop. For example:

Gain = (profilePercent) * (1 - 1 / parallelPaths)
   where
5   profilePercent = percent of runtime spent in this loop
     parallelPaths = number of paths which can be executed in parallel

The amount of ASIC area required to implement a source code loop is determined by summing the individual areas of all its mapped cells and estimating the
10 additional area required to interconnect the cells. The size of the cells and their interconnect depends on the number bits needed to implement the required data precision. The ASIC area can serve as a figure of merit for the cost of each loop. For example:

Cost = cellArea + MAX(0, (interconnectArea - overTheCellArea))
15   where
   cellArea = sum of all component cell areas
   overTheCellArea = cellArea * (per cell area available for interconnects)
     interconnectArea = (number of interconnects) *
          (interconnectLength) * (interconnect width)
20   interconnectLength = (square root of the number of cells) / 3

For further information on estimating interconnect area see B. Preas, M. Lorenzetti; Physical Design Automation of VLSI Systems; Benjamin/Cummings Publishing Company; 1988; pp. 31-64.

25   The method does not actually calculate the figures of merit for all the loops in the source code. The compiler is given two runtime parameters: the maximum area for a single ASIC block, and the maximum total ASIC area available, depending on the targeted runtime system. It first sorts the loops in descending order of their percentage of runtime, and then estimates the figures of merit for each loop until it reaches a
30 predetermined limit in the total amount of area estimated. The predetermined limit is a constant times the maximum total ASIC area available. Loops that require an area larger than a single ASIC block may be skipped for a simpler implementation. Finally, with all the loops for which figures of merit have been calculated, a knapsack algorithm is applied to select the loops. This procedure can be trivially extended to handle the

case of targeting multiple ASICs if there is no gain or cost associated with being in different ASICs. For a general discussion of knapsack algorithms see Syslo, Deo, Kowalik; Discrete Optimization Algorithms; Prentice-Hall; 1983; pp. 118-176.

The various source code loops which are packed onto a single ASIC are

5   generally independent of each other. With certain types of ASICs, namely a field programmable gate array (FPGA), it is possible to change at runtime some or all of the functions on the FPGA. The FPGA has one or more independent blocks of reconfigurable logic. Each block may be reconfigured without affecting any other block. Changing which functions are currently implemented may be desirable as the

10   computer program executes different areas of code, or when an entirely different computer program is loaded, or when the amount of available FPGA logic changes.

A reconfigurable FPGA environment presents the following problems for the compiler to solve: selecting the total set of functions to be implemented, partitioning the functions across multiple FPGA blocks, and scheduling the loading and activation

15   of FPGA blocks during the program execution. These problems cannot be solved optimally in polynomial time. The following paragraphs describe some heuristics which can be successfully applied to these problems.

The set of configurations simultaneously coexisting on an FPGA at a single instant of time will be referred to as a snapshot. The various functions comprising a

20   snapshot are partitioned into the separate blocks by the compiler in order to minimize the block's stall time and therefore minimize the overall execution schedule. A block will be stalled if the microprocessor has issued a new activate_configuration instruction, but all the functions of the previous configuration have not yet completed. The partitioning will group together functions that finish at close to the same time. All

25   the functions which have been selected by the knapsack algorithm are sorted according to their ideal scheduled finish times (the ideal finish times assume that the blocks have been downloaded and activated without delay so that the functions can be invoked at their scheduled start times). Traversing the list by increasing finish times, each function is assigned to the same FPGA block until the FPGA block's area capacity is

30   reached. When an FPGA block is filled, the next FPGA block is opened. After all

functions have been assigned to FPGA blocks, the difference between the earliest and the latest finish times is calculated for each FPGA block. Then each function is revisited in reverse (decreasing) order. If reassigning the function to the next FPGA block does not exceed its area capacity and reduces the maximum of the two

5   differences for the two FPGA blocks, then the function is reassigned to the next FPGA block.

After the functions are partitioned, each configuration of an FPGA block may be viewed as a single task. Its data and control dependencies are the union of its assigned function's dependencies, and its required time is the difference between the

10 latest finish time and the earliest start time of its assigned functions. The set of all such configuration tasks across all snapshots may be scheduled with standard multiprocessor scheduling algorithms, treating each physical FPGA block as a processor. This will schedule all the activate_configuration instructions.

A common scheduling algorithm is called list scheduling. In list scheduling, the

15 following steps are a typical implementation:

1.   Each node in the task graph is assigned a priority. The priority is defined as the length of the longest path from the starting point of the task graph to the node. A priority queue is initialized for ready tasks by inserting every task that has no immediate predecessors. Tasks are sorted in decreasing order of task priorities.

20   2.   As long as the priority queue is not empty do the following:

a.   A task is obtained from the front of the queue.

b.   An idle processor is selected to run the task.

c.   When all the immediate predecessors of a particular task are executed, that successor is now ready and can be inserted into the priority queue.

25   For further information on multiprocessor scheduling algorithms see A. Zomaya; Parallel and Distributed Computing Handbook; McGraw-Hill; 1996; pp. 239-273.

All the load_configuration instructions may be issued at the beginning of the program if the total number of configurations for any FPGA block does not exceed the

30 capacity of the FPGA block's configuration cache. Similarly, the program may be

divided into more than one section, where the total number of configurations for any

FPGA block does not exceed the capacity of the FPGA block's configuration cache.

Alternatively, the load_configuration instructions may be scheduled at the lowest

preceding branch point in the program's control flow graph which covers all the

5   block's activate_configuration instructions. This will be referred to as a covering load

instruction. This is a preliminary schedule for the load instructions, but will lead to

stalls if the actual load time exceeds the time the microprocessor requires to go from

the load_configuration instruction to the first activate_configuration instruction. In

addition, the number of configurations for an FPGA block may still exceed the capacity

10   of its configuration cache. This will again lead to stalls in the schedule. In such a

case, the compiler will compare the length of the stall versus the estimated gains for

each of the configurations in contention. The gain of a configuration is estimated as

the sum of the gains of its assigned functions. Among all the configurations in

contention, the one with the minimum estimated gain is found. If the stall is greater

15   than the minimum gain, the configuration with the minimum gain will not be used at

that point in the schedule.

When a covering load instruction is de-scheduled as above, tentative

load_configuration tasks will be created just before each activate_configuration

instruction. These will be created at the lowest branch point immediately preceding the

20   activate instruction. These will be referred to as single load instructions. A new

attempt will be made to schedule the single load command without exceeding the FPGA

block's configuration cache capacity at that point in the schedule. Similarly to the

previous scheduling attempt, if the number of configurations again exceeds the

configuration cache capacity, the length of the stall will be compared to the estimated

25   gains. In this case, however, the estimated gain of the configuration is just the gain of

the single function which will be invoked down this branch. Again, if the stall is

greater than the minimum gain, the configuration with the minimum gain will not be

used at that point in the schedule.

If a de-scheduled load instruction is a covering load instruction, the process will

30   recurse; otherwise if it is a single load instruction, the process terminates. This process

can be generalized to shifting the load instructions down the control flow graph one
step at a time and decreasing the number of invocations it must support. For a single
step, partition each of the contending configurations into two new tasks. For the
configurations which have already been scheduled, split the assigned functions into

5   those which finish by the current time and those that don't. For the configuration
which has not been scheduled yet, split the assigned functions into those which start
after the stall time and those that don't.

Branch prediction may be used to predict the likely outcome of a branch and to
load in advance of the branch a configuration likely to be needed as a result of the

10   branch. Inevitably, branch prediction will sometimes be unsuccessful, with the result
that a configuration will have been loaded that is not actually needed. To provide for
these instances, instructions may be inserted after the branch instruction to clear the
configuration loaded prior to the branch and to load a different configuration needed
following the branch, provided that a net execution-time savings results.

15   It will be appreciated by those of ordinary skill in the art that the invention can
be embodied in other specific forms without departing from the spirit or essential
character thereof. The presently disclosed embodiments are therefore considered in all
respects to be illustrative and not restrictive. The scope of the invention is indicated by
the appended claims rather than the foregoing description, and all changes which come

20   within the meaning and range of equivalents thereof are intended to be embraced
therein.

Claims:

1.   A computer implemented method for the automatic compilation of a
computer program written in a high level programming language into a program for
execution by one or more application specific integrated circuits coupled with a
5   microprocessor, the method comprising the steps of:

        automatically determining a set of one or more special instructions, to be
executed by said one or more application specific integrated circuits, that will result in
a relative performance improvement for a given input computer program written for
execution by the microprocessor; and

10   generating code including said one or more special instructions.

2.   The method of Claim 1, wherein generating code comprises producing
code for execution by the microprocessor, including instructions for loading and
activating said functions.

15

3.   The method of Claim 2, comprising the further steps of:
        selecting code blocks the functions of which are to be performed by circuitry
within the one or more application specific integrated circuits;
        grouping the code blocks into groups based on at least one of an area constraint
20   and an execution timing constraint;
        scheduling loading of said functions; and
        scheduling activation of said functions.

4.   The method of Claim 2, comprising the further step of producing
25   detailed integrated circuit layouts of said circuitry.

5.   The method of Claim 4, comprising the further step of producing
configuration data for said functions.

6.   The method of Claim 2, wherein said instructions include special instructions to load, activate, invoke and/or release functions implemented on an application specific integrated circuit.

5   7.   The method of Claim 2, wherein grouping comprises calculating start and finish times for the selected blocks of code.

8.   The method of Claim 7, wherein the start and finish times are calculated assuming that the selected code blocks are implemented in parallel with a fixed
10   overhead for each parallel operation.

9.   The method of Claim 8, wherein the fixed overhead is calculated as OV = I + A + L, where I is an average time required to invoke the application specific integrated circuit as a coprocessor instruction; A is an average time required to issue an
15   activate configuration instructions plus an average stall time for activation; and L is an average time required to issue a load configuration instruction plus an average stall time for loading.

10.   The method of Claim 7, wherein grouping is performed such that a
20   difference between the latest and earliest finish times within a group is minimized.

11.   The method of Claim 7, wherein grouping is performed such that for each group, circuitry for performing the functions of that group does not exceed a specified capacity of a block of an application integrated circuit.

25
12.   The method of Claim 7, wherein grouping further comprises:
        opening a new group with a total assigned area of zero;
        sorting and traversing the code blocks in a predetermined order;
        for each code block, if the area of the block plus the group's assigned area does
30   not exceed a specified maximum area for a single group, adding the code block to the

group and adding the area of the code block to the group's assigned area; otherwise, opening a new group, adding the code block to the new group and adding the area of the code block to the new group's assigned area.

5   13.   The method of Claim 12, wherein said predetermined order is in increasing order of finish times as a primary key, and increasing order of start times as a secondary key.

      14.   The method of Claim 13, wherein grouping comprises the further steps

10   of:

      traversing the code blocks in decreasing order of finish times;

      for each code block, determining a start spread and finish spread of a group to which the code block belongs, wherein the start spread is the difference between the latest and earliest start times of all of the code blocks belonging to the same group, and

15   the finish spread is the difference between the latest and earliest finish times of all of the code blocks belonging to the same group; and

      reassigning the code block to a different group if the code block's area plus the different group's assigned area does not exceed the specified maximum area for a single group, and if reassigning the code block results in a net improvement in at least one of

20   start spread and finish spread for the group to which the code block belongs and the different group.

      15.   The method of Claim 2, wherein selecting comprises sampling the percentage of time spent in each block of code when the computer program is executed

25   on a single microprocessor.

      16.   The method of Claim 15, wherein selecting further comprises:

      parsing the high level programming language into an intermediate data structure representing control and data dependencies of the computer program; and

30   analyzing the amount of implicit parallelism in the intermediate data structure.

17.   The method of Claim 16, wherein selecting further comprises, for at least some of the code blocks of the computer program, estimating the cost and benefit of implementing a code block using circuitry within an application specific integrated circuit.

5

18.   The method of Claim 17, wherein estimating the cost and benefit of implementing a code block comprises:

estimating a reduction in execution time if the code block is implemented as an application specific integrated circuit; and

10   estimating a layout area required if the code block is implemented as an application specific integrated circuit.

19.   The method of Claim 18, wherein selecting further comprises:

accepting a first runtime parameter representing a maximum area of a single

15   block of an application specific integrated circuit and a second runtime parameter representing a maximum total area for all blocks to be considered for implementation as application specific integrated circuits; and

selecting a set of code blocks which satisfies the first and second runtime parameters and which maximizes a total estimated reduction in execution time.

20

20.   The method of Claim 19, wherein selecting a set of code blocks which satisfies the first and second runtime parameters and which maximizes a total estimated reduction in execution time comprises:

sorting and traversing the code blocks in decreasing order of reduction in

25   execution time; and

for each code block:

if the reduction equals zero, terminate;

estimate the required layout area;

if the area exceeds the specified maximum area for a single block of an

30   application specific integrated circuit, skip this code block;

multiplying the specified maximum total area for all blocks by a constant greater than one;

if a total area of previously selected code blocks plus an estimated required layout area for a current code block exceeds the specified maximum total

5   multiplied by the constant, terminate;

otherwise, select the code block; and

using a knapsack algorithm and the maximum total area to perform a further selection on the selected code blocks.

10   21.   The method of Claim 18, wherein the reduction in execution time is estimated in accordance with the formula $R = T(1 - 1/P)$ where T is a percentage of execution time spent in the code block and P is a number of paths which can be executed in parallel in the code block.

15   22.   The method of Claim 18, wherein the intermediate data structure is a tree structure containing nodes, and estimating the layout area comprises:

performing bottom-up traversal of the tree structure;

mapping each node in the tree to a cell from a library of circuit primitives;

calculating a total area of the mapped cells; and

20   calculating an additional area required for cell interconnections.

23.   The method of Claim 22, wherein mapping is performed in accordance with multiple predetermined mappings including at least one of the following: scalar variables map to registers; arrays map to register files; addition and subtraction

25   operators map to adders; increment and decrement operators map to adders; multiplications and division operators map to multipliers; equality and inequality operators map to comparators; $+=, -=$ operators map to accumulators; $*=, /=$ operators map to multiply-accumulators, $<<, >>$ operators map to shift registers; &, |, ¯, ˆ operators map to boolean gates, branches map to a state machine, and loops

30   map to a state machine.

24.   The method of Claim 22, wherein mapping includes determining a number of significant bits required to support a data precision expected by the computer program.

5   25.   The method of Claim 22, wherein calculating an additional area required for interconnections is performed in accordance with the following formula: area = max(0, (A - B)) where A is an estimate of total area required for interconnections and B is an estimate of area available within the mapped cells for use by interconnections.

10   26.   The method of Claim 25, wherein A is calculated as the product of a runtime parameter for the width of an interconnection, an average length of an interconnection calculated as a fraction times the square root of the number of mapped cells, and the total number of interconnections.

15   27.   The method of Claim 25, wherein B is calculated as the product of a runtime parameter for the fraction of cell area for interconnections and the total area of all of the mapped cells.

        28.   The method of Claim 16, comprising the further step of estimating a
20   reduction in execution time for each group.

        29.   The method of Claim 28, wherein scheduling activation is performed such that overall execution time is minimized subject to at least one of an area constraint and an execution time constraint.

25

        30.   The method of Claim 29, wherein scheduling activation is performed such that data and control dependencies of all code blocks within a group are not violated.

31.   The method of Claim 29, wherein scheduling activation is performed such that a specified number of simultaneous blocks of an application specific circuit is not exceeded.

5   32.   The method of Claim 29, wherein scheduling further comprises:

modeling each group as a separate task;

modeling as a processor each available block of reconfigurable logic on an application specific integrated circuit; and

running a modified multiprocessor scheduling algorithm.

10

33.   The method of Claim 32, wherein the intermediate data structure is a graph in which arcs represent dependencies, and wherein modeling each group as a separate task comprises:

for each group, adding a node to the graph;

15   for each code block assigned to a group, modifying the graph such that arcs that previously pointed to the code block point instead to a node representing the group;

determining a difference between a latest finish time and an earliest start time of code blocks assigned to the group; and

setting a required time of the group equal to said difference.

20

34.   The method of Claim 32, wherein running a modified multiprocessor scheduling algorithm comprises:

running a standard list scheduling multiprocessor scheduling algorithm;

during running of the algorithm, in the event no processor is available when a

25   newly-ready task becomes ready:

calculating a stall time until a processor would become available;

create a list of contending tasks including the newly-ready task and tasks scheduled to be executing at a time the newly-ready task becomes ready; and

finding a contending task with a minimum estimated reduction in execution

30   time.

35.   The method of Claim 34, wherein running the modified multiprocessor scheduling algorithm further comprises:

if the stall time is less than or equal to the minimum reduction, scheduling the newly-ready task to execute when a processor becomes available and continuing to run

5   the multiprocessor scheduling algorithm.

36.   The method of Claim 35, wherein running the modified multiprocessor scheduling algorithm further comprises, if the stall time is greater than the minimum reduction, discarding the task with the minimum reduction and continuing to run the

10   multiprocessor scheduling algorithm.

37.   The method of Claim 35, wherein running the modified multiprocessor scheduling algorithm further comprises, if the stall time is greater than the minimum reduction:

15   replacing the newly-ready task with two new tasks, a first new task containing code blocks of the newly-ready task having start times later than when a processor would become available, and a second new task containing other code blocks of the newly-ready task;

replacing respective tasks scheduled to be executing at a time the newly-ready

20   task becomes ready with two new respective tasks, a first new task containing code blocks of the newly-ready task having start times later than when a processor would become available, and a second new task containing other code blocks of the newly-ready task.

25   38.   The method of Claim 37, wherein running the modified multiprocessor scheduling algorithm further comprises:

of the new tasks, finding a task with a minimum reduction in execution time; and

discarding the task with the minimum reduction.

30

39.   The method of Claim 28, wherein scheduling loading is performed such that overall execution time is minimized subject to at least one of an area constraint and an execution time constraint.

5   40.   The method of Claim 39, wherein scheduling loading is performed such that each function activation is preceded by loading.

41.   The method of Claim 39, wherein scheduling loading is performed such that a specified capacity for coexisting groups loaded for a block of an application

10   specific circuit is not exceeded.

42.   The method of Claim 39, wherein the data structure includes a control flow graph, and wherein scheduling loading comprises:
    modeling each group as a task and each available block of an application

15   specific integrated circuit as a processor with a specified maximum number of simultaneous tasks;
    for each group activation of which has been successfully scheduled, creating a new load_group task having a finish time equal to a finish time of a task representing the group and having a start time equal to a start time of the task representing the group

20   minus a runtime parameter specifying a time required to load a group.

43.   The method of Claim 42, wherein scheduling loading further comprises, for each new load_group task, inserting a node into the control flow graph.

25   44.   The method of Claim 42, wherein scheduling loading further comprises:
    finding a branching node in the control flow graph immediately preceding activation;
    calculating a stall time of a load_group task as a finish time of the branching node minus the load_group task start time;

if the stall time is less than or equal to the estimated reduction in execution time for the group, creating a control flow arc from the branching node to the load_group task;

otherwise, discarding the load_group task and discarding the group.

5

45.   The method of Claim 43 or Claim 44 wherein scheduling loading further comprises running a modified list processing multiprocessor scheduling algorithm.

46.   The method of Claim 45, wherein running a modified list processing
10   multiprocessor scheduling algorithm comprises:

running a list scheduling multiprocessor scheduling algorithm with a specified maximum number of simultaneous tasks per processor;

during running of the algorithm, in the event no processor is available when a newly-ready task becomes ready:

15   calculating a stall time until a processor would become available;

create a list of contending tasks including the newly-ready task and tasks scheduled to be executing at a time the newly-ready task becomes ready; and

finding a contending task with a minimum estimated reduction in execution

time.

20

47.   The method of Claim 46, wherein running the modified multiprocessor scheduling algorithm further comprises:

if the stall time is less than or equal to the minimum reduction, scheduling the newly-ready task to execute when a processor becomes available, adjusting the schedule
25   for a corresponding group task and continuing to run the multiprocessor scheduling algorithm.

48.   The method of Claim 46, wherein running the modified multiprocessor scheduling algorithm further comprises, if the stall time is greater than the minimum

reduction, discarding the task with the minimum reduction and its corresponding group and continuing to run the multiprocessor scheduling algorithm.

49. The method of Claim 48, wherein running the modified multiprocessor
5   scheduling algorithm further comprises, if in the control flow graph a branching node intervenes between a node representing a discarded load_group task and a node representing activation of the corresponding group:

finding a branching node in the control flow graph immediately preceding activation;

10   calculating a stall time of a load_group task as a finish time of the branching node minus the load_group task start time;

if the stall time is less than or equal to the estimated reduction in execution time for the group, creating a control flow arc from the branching node to the load_group task;

15   otherwise, discarding the load_group task and discarding the group.

50. The method of Claim 48, wherein running the modified multiprocessor scheduling algorithm further comprises, if the stall time is greater than the minimum reduction:

20   replacing the newly-ready task with two new tasks and corresponding groups, a first new group containing code blocks of the newly-ready task having start times later than when a processor would become available, and a second new group containing other code blocks of the newly-ready task;

replacing respective tasks scheduled to be executing at a time the newly-ready
25   task becomes ready with two new respective tasks and corresponding groups, a first new group containing code blocks of the newly-ready task having start times later than when a processor would become available, and a second new group containing other code blocks of the newly-ready task.

51.   The method of Claim 50, wherein running the modified multiprocessor scheduling algorithm further comprises:

of the new tasks, finding a task with corresponding group having a minimum reduction in execution time; and

5   discarding the task with the corresponding partition having the minimum reduction.

52.   The method of Claim 51, wherein running the modified multiprocessor scheduling algorithm further comprises, if in the control flow graph a branching node

10   intervenes between a node representing a discarded load_group task and a node representing activation of the corresponding group:

finding a branching node in the control flow graph immediately preceding activation;

calculating a stall time of a load_group task as a finish time of the branching

15   node minus the load_group task start time;

if the stall time is less than or equal to the estimated reduction in execution time for the group, creating a control flow arc from the branching node to the load_group task;

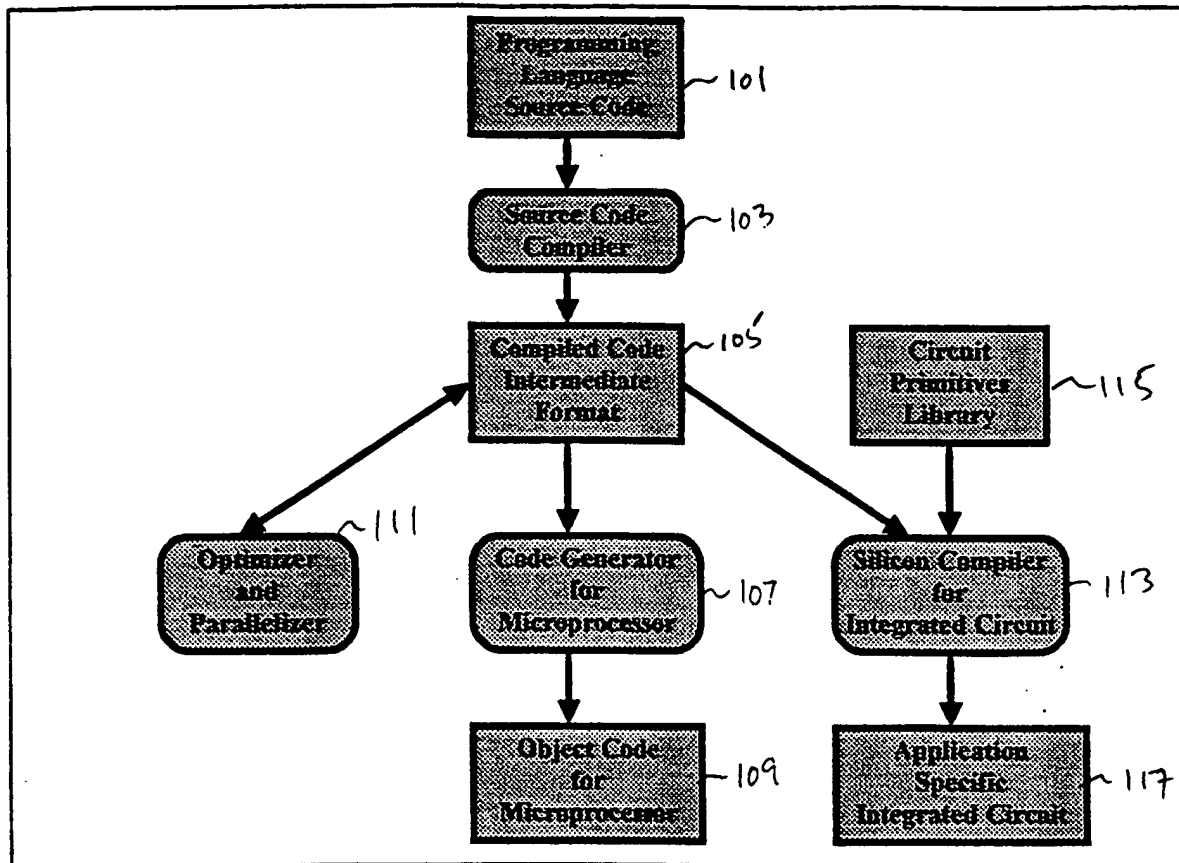otherwise, discarding the load_group task and discarding the group.

20

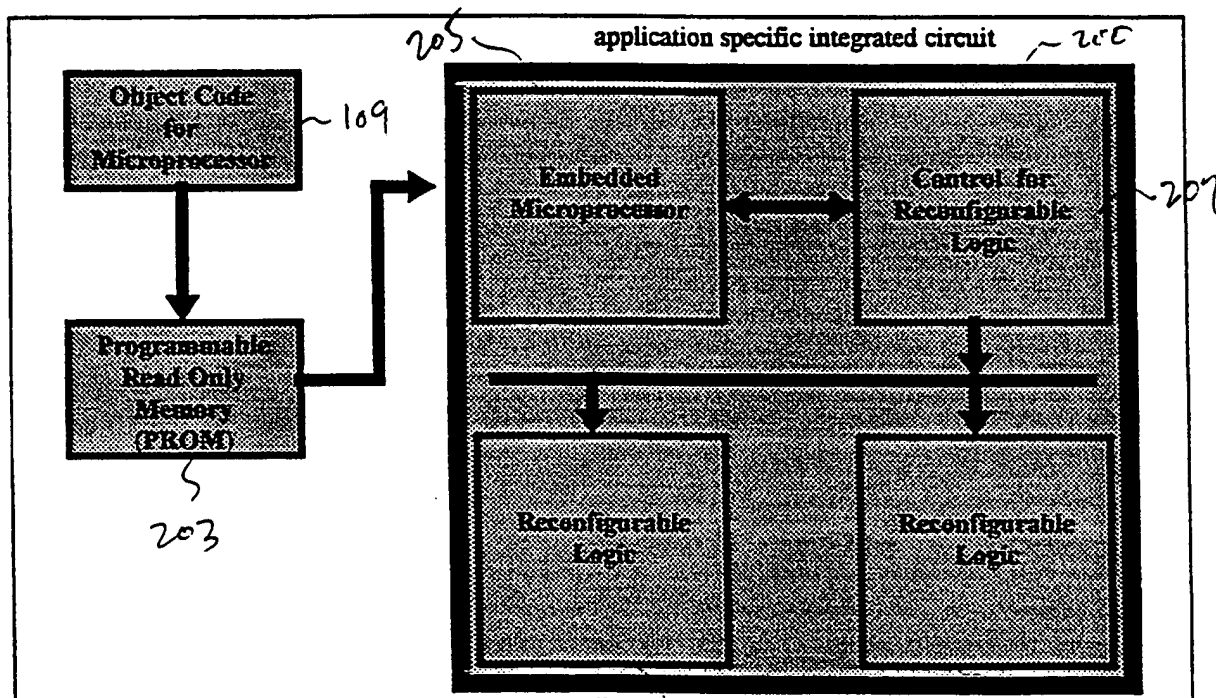Figure 1: design methodology flow diagram

Figure 2: application specific integrated circuit operation

```
/* finite impulse response filter */
for (j = 0; j < SAMPLES; j++)
{
    out[j] = 0;

    /* inner loop */
    for (i = 0; i < FILTER_TAPS; i++)
    {
        out[j] += coeff[i] * sample[j + i];
    }
}
```

Figure 3: high level source code example

mult	coeff[registerA], sample[registerB], registerC
add	registerC, registerD
add	registerA, 1, registerA
add	registerB, 1, registerB

Figure 4: inner loop compiled for sequential standard microprocessor object code
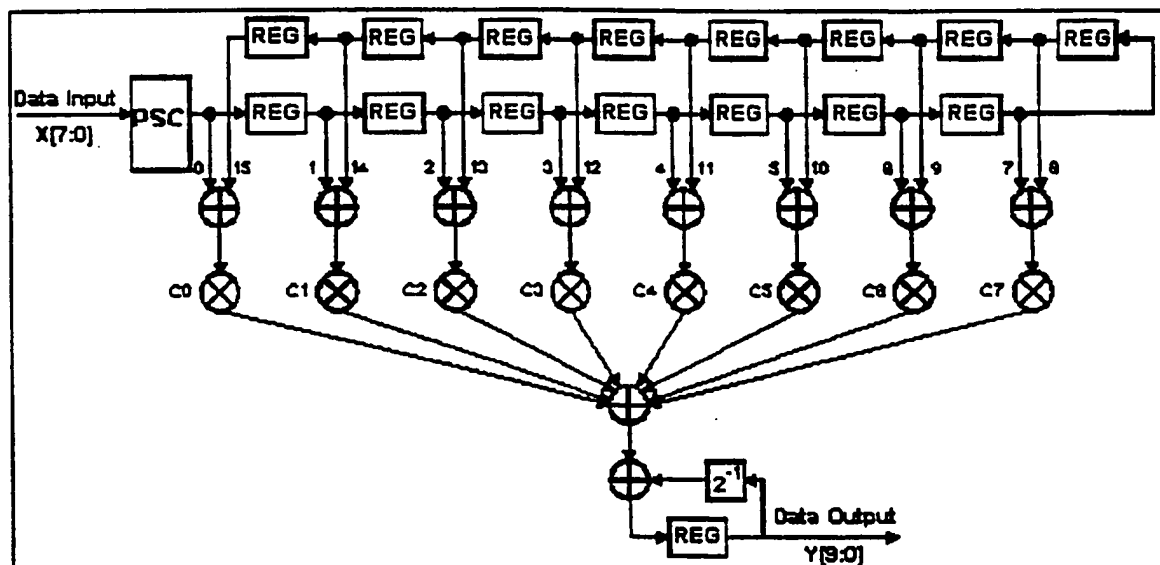

Figure 5: inner loop compiled for parallel execution on application specific circuitry

INTERNATIONAL SEARCH REPORT

	International application No.
	PCT/US98/13563

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :GO6F 12/02
US CL :395/709

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/709, 704

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	USA 5,794,044 (YELLIN) 11 AUGUST 1998, COL. 1-10	1,2,4,5-7, 11
X	USA 5,625,797 (FERRY ET AL.) 29 APRIL 1997, COL. 1-6,	1,2,4,5-7, 11

☐ Further documents are listed in the continuation of Box C.   ☐ See patent family annex.

	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance		
"E"	earlier document published on or after the international filing date	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed	"&"	document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
22 SEPTEMBER 1998	22 OCT 1998

Name and mailing address of the ISA/US	Authorized officer
Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231	JOHN Q. CHAVIS
Facsimile No. (703) 305-3230	Telephone No. (703) 305-9600

Form PCT/ISA/210 (second sheet)(July 1992)*